Theoretical Computer Science 411 (2010) 4333–4356

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL☆

Filip Marić*

Faculty of Mathematics, University of Belgrade, Belgrade, Serbia

ARTICLE INFO

Article history:
Received 12 December 2008
Received in revised form 16 September 2010
Accepted 16 September 2010
Communicated by D. Sannella

Keywords:
Formal program verification
SAT problem
DPLL procedure
Isabelle

ABSTRACT

We present a formalization and a formal total correctness proof of a MiniSAT-like SAT solver within the system Isabelle/HOL. The solver is based on the DPLL procedure and employs most state-of-the-art SAT solving techniques, including the conflict-guided backjumping, clause learning, and the two-watched unit propagation scheme. A shallow embedding into Isabelle/HOL is used and the solver is expressed as a set of recursive HOL functions. Based on this specification, the Isabelle's built-in code generator can be used to generate executable code in several supported functional languages (Haskell, SML, and OCaml). The SAT solver implemented in this way is, to our knowledge, the first fully formally and mechanically verified modern SAT solver.

1. Introduction

The propositional satisfiability problem (SAT) is the problem of deciding if there is a truth assignment under which a given propositional formula (in conjunctive normal form) evaluates to true. It is a canonical NP-complete problem [3] and it holds a central position in the field of computational complexity. The SAT problem is also important in many practical applications such as electronic design automation, software and hardware verification, artificial intelligence, and operations research. Thanks to recent advances in propositional solving technology, SAT solvers are becoming the tool for attacking more and more practical problems. Most modern SAT solvers are based on the *Davis–Putnam–Logemann–Loveland (DPLL) procedure* [5,4] and its modifications.

Since SAT solver are used in applications that are very sensitive (e.g., software and hardware verification), their misbehavior could be both financially expensive and dangerous from the aspect of security. Clearly, having a trusted SAT solving system is vital. This can be achieved in two different ways.

1. One approach is to extend an *online* SAT solver with the possibility of generating models of satisfiable formulas and proofs of unsatisfiability for unsatisfiable formulas. The generated models and proofs are then checked *offline* by an independent trusted checker [26,7].
2. Another approach is to apply software verification techniques and verify the implementation of the SAT solver itself, so that it becomes trusted [13,23,15].

The first approach has successfully been used in recent years. It is relatively easy to implement, but it has some drawbacks. Generating object-level proofs introduces about 10% overhead to the solver's running time and proof checking can also take a significant amount of time [7]. More importantly, since proofs are very large objects, they can consume up to several gigabytes of storage space. Since proof checkers have to be trusted, they must be very simple programs so that they could

☆ This work was partially supported by Serbian Ministry of Science grant 144030.
* Tel.: +381 11 2027863.
E-mail address: filip@matf.bg.ac.rs.

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.09.014

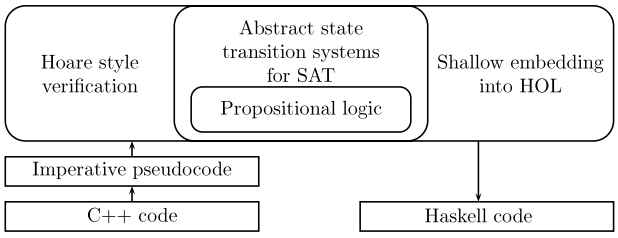

Fig. 1. SAT verification project.

be "verified" only by manually inspecting their source code [7]. On the other hand, in order to handle large proof objects, checkers must use specialized functionality of the underlying operating system, which reduces the level of their confidence.[1]

In this work we take the second, harder, approach and formally verify a full implementation of a SAT solver. There are several reasons for doing this.

1. We believe that this verification effort could help in a better theoretical understanding of how and why modern SAT solver procedures work.
2. Verified SAT solvers can serve as the trusted kernel checkers for verifying results of other untrusted verifiers such as BDDs, model checkers, and SMT solvers [23]. Also, verification of some SAT solver modules (e.g., Boolean constraint propagation) can serve as a basis for creating a verified, yet efficient, proof checker for SAT.
3. The overheads of generating and storing unsatisfiability proofs can be avoided if the SAT solver itself is trusted.
4. We hope that this work contributes to the *Verification Grand Challenge* [25], and adds to the growing collection of non-trivial software that has been fully formally verified.

In order to prove the correctness of a SAT solver implementation, it needs to be formalized in some meta-theory so its properties can be analyzed by using an appropriate mathematical apparatus. In order to achieve the desired, highest level of trust, formalization in a classical "pen-and-paper" fashion is not satisfactory and a mechanized and machine-checkable formalization is required.

Results presented in this paper constitute a significant part of our SAT verification project [17], illustrated in Fig. 1. All formalizations done within the project were made within the system Isabelle/HOL [21].[2] As a part of this project, abstract state transition systems for SAT [11,20] have been formalized. Following these formalizations, we have implemented a modern SAT solver ArgoSAT[3] in C++. Since formal verification of the real C++ code was beyond our reach, we have developed a corresponding SAT solver description in an *imperative pseudo-language*, within a tutorial on the modern SAT solving technology [15]. This description (obtained from an executable SAT solver) was semi-mechanically verified using Hoare logic (verification conditions were manually generated and then verified within Isabelle/HOL) [15].

In the current paper, a different approach is pursued. A shallow embedding into HOL is used, i.e., the SAT solver is expressed as a set of recursive functions in HOL (which is, for this purpose, treated as a *pure functional* programming language). From this specification, an executable SAT solver in several functional languages (e.g., Haskell, SML, OCaml) can be automatically extracted. The extracted solver achieves a much higher level of trust, since the whole formalization is done within the theorem prover. In addition to this important conceptual difference, this paper also brings a formal proof of termination of a modern solver, not previously given.

In the rest of the paper, a full, self-contained, implementation of a SAT solver within Isabelle/HOL will be presented. However, some familiarity with modern SAT solving technology is assumed (the reader can consult tutorials given in the literature [1,15,8,6]).

Overview of the paper. The rest of the paper is structured as follows. In Section 2 we give some background information about the DPLL procedure and its modifications. We also give some background on program verification. In Section 3 we introduce basic notions of the system Isabelle and formulate an underlying theory for our formalization. The central section of the paper is Section 4 in which we present the specification of the SAT solver and introduce correctness conditions along the way. In Section 5 we outline the correctness proof of our implementation and in Section 6 we discuss some aspects of the proof management. In Section 7 we list some of the related work, in Section 8 we list some possible directions for further work, and in Section 9 we draw final conclusions.

2. Background

DPLL procedure and its modifications. Most modern SAT solvers are based on the *Davis–Putnam–Logemann–Loveland (DPLL) procedure*. Its original recursive version is shown in Fig. 2, where F denotes a set of propositional clauses tested for satisfiability and $F[l \to \top]$ denotes the formula obtained from F by substituting the literal l with $\top$, its opposite literal $\bar{l}$

[1] For example, the proof checker used in SAT competitions uses Linux's `mmap` functionality [7].

[2] The original proof documents are available online [14].

[3] The web page of ArgoSAT is http://argo.matf.bg.ac.rs.

```
function dpll (F : Formula) : (SAT, UNSAT)
begin
   if F is empty then
      return SAT
   else if there is an empty clause in F then
      return UNSAT
   else if there is a pure literal l in F then
      return dpll(F[l → ⊤])
   else there is a unit clause [l] in F then
      return dpll(F[l → ⊤])
   else begin
      select a literal l occurring in F
      if dpll(F[l → ⊤]) = SAT then
         return SAT
      else
         return dpll(F[l → ⊥])
   end
end
```

Fig. 2. DPLL algorithm — recursive definition.

with $\bot$, and simplifying afterwards. A literal is *pure* if it occurs in the formula but its opposite literal does not occur. A clause is *unit* if it contains only one literal. This recursive implementation is practically unusable for larger formulae and therefore it is not used in modern SAT solvers, nor in this paper.

Starting with the work on the GRASP and SATO systems [19,24] and continuing with Chaff, BerkMin and MiniSAT [18,9,6], the spectacular improvements in the performance of DPLL-based SAT solvers achieved in the last years are due to (i) several conceptual enhancements of the original DPLL procedure, such as *backjumping* (a form of non-chronological backtracking), *conflict-driven lemma learning*, and *restarts*, (ii) advanced heuristic components (e.g., literal selection strategies) and (iii) better implementation techniques, such as the *two-watched literals* scheme for unit propagation. These advances make it possible to decide satisfiability of some industrial SAT problems with tens of thousands of variables and millions of clauses.

Abstract state transition systems for SAT. During the last few years two state transition systems which model modern DPLL-based SAT solvers and related SMT solvers have been published [20,11]. These descriptions define the top-level architecture of solvers as a mathematical object that can be grasped as a whole and fruitfully reasoned about. Both systems are accompanied by pen-and-paper correctness and termination proofs. Although they succinctly and accurately capture all major aspects of the solvers' global operation, they are still high level and far from the actual implementations. Both systems model the solver behavior as transitions between states that represent the values of global variables of the solver. These include the set of clauses F and the corresponding assertion trail M. Transitions between states are performed only by using precisely defined transition rules. The solving process is finished when no transition rule applies and the final state is reached.

The system of Nieuwenhuis et al. [20] is very coarse. It can capture many different strategies seen in the state-of-the-art SAT solvers, but this comes at a price. Several important aspects still have to be specified in order to build the implementation based on the given set of rules.

The system of Krstic and Goel [11] gives a more detailed description of some parts of the solving process (particularly the conflict analysis phase) than the previous one. Since this system is used as a basis of the implementation given in this paper, we list its transition rules in Fig. 3. Together with the formula F and the trail M, the state of the solver is characterized by the conflict analysis set C which is either the set of literals or the distinguished symbol *no_cflct*. The input to the system is an arbitrary set of clauses F_0, modeled as the initial state in which $F = F_0$, $M = [\,]$, and C=*no_cflct*. The rules have guarded assignment form: above the line is the condition that enables the application of the rule, below the line is the update to the state variables.

Formal program verification. Formal program verification is the process of proving that a computer program meets its specification which formally describes the expected program behavior. Early results date back to the 1950s and pioneers in this field were A. Turing, J. von Neumann and J. McCarthy. In the late 1960s R. Floyd introduced equational reasoning on flowcharts for proving program correctness and T. Hoare introduced axiomatic semantics for programming constructs. Following the lessons from major software failures in recent years, an increasing amount of effort is being invested in this field.

To achieve the highest level of trust, mechanically checkable formal proofs of correctness are required. Many fundamental algorithms and properties of data structures have been formalized and verified in this way. Also, a lot of work has been devoted to formalization of programming language semantics, compilers, communication protocols, security protocols, etc. Many of the early results in mechanical program verification were carried out by Boyer and Moore using their theorem prover. Theorem provers that are most commonly used for program verification nowadays are Isabelle, HOL, Coq, PVS, Nuprl, etc. A large collection of formalized theories (of both pure mathematics and computer science) mechanically checked by the theorem prover Isabelle is available in *Archive of formal proofs* (http://afp.sourceforge.net).

Decide:
$$\frac{l \in F \qquad l, \bar{l} \notin M}{M := M\, l^d}$$
UnitPropag:
$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \qquad \bar{l}_1, \ldots, \bar{l}_k \in M \qquad l, \bar{l} \notin M}{M := M\, l}$$
Conflict:
$$\frac{C = no\_cflct \qquad \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad l_1, \ldots, l_k \in M}{C := \{l_1, \ldots, l_k\}}$$
Explain:
$$\frac{l \in C \qquad l \vee \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad l_1, \ldots, l_k \prec l}{C := C \cup \{l_1, \ldots, l_k\} \setminus \{l\}}$$
Learn:
$$\frac{C = \{l_1, \ldots, l_k\} \qquad \bar{l}_1 \vee \ldots \vee \bar{l}_k \notin F}{F := F \cup \{\bar{l}_1 \vee \ldots \vee \bar{l}_k\}}$$
Backjump:
$$\frac{C = \{l, l_1, \ldots, l_k\} \qquad \bar{l} \vee \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad \text{level } l > m \geq \text{level } l_i}{C := no\_cflct \qquad M := M^{[m]}\, \bar{l}}$$
Forget:
$$\frac{C = no\_cflct \qquad c \in F \qquad F \setminus c \vDash c}{F := F \setminus c}$$
Restart:
$$\frac{C = no\_cflct}{M := M^{[0]}}$$

Fig. 3. Rules of DPLL as given by Krstić and Goel [11].

Formal program verification by shallow embedding into HOL. Shallow embedding into higher-order logic is a technique that is widely used for verification, despite its well-known limitations [2]. This success is due in part to the simplicity of the approach: a formal model of the operational or denotational semantics of the language is not required and many technical difficulties (e.g., the representation of binders) are avoided altogether. Furthermore, the proof methods used are mainly standard induction principles and equational reasoning, and no specialized program logic (e.g., Hoare logic) is necessary. The specifications may be turned into executable code directly by means of code generation [10]. The main drawback of this approach is that all programs must be expressed as purely functional. As the notion of side-effect is alien to the world of HOL functions, programs with imperative updates of references or arrays cannot be expressed directly which heavily effects the efficiency of the generated code. Still, approaches to overcome these difficulties have been proposed recently [2].

3. Underlying theory

In order to create and reason about the correctness of a SAT solver, we have to formally define some basic notions of propositional logic. The full formalization has been made in the higher-order logic of the system Isabelle and basic knowledge about this system is assumed in the rest of the paper. We will use a syntax similar to the syntax used in Isabelle/HOL. Formulas and logical connectives of this logic ($\wedge$, $\vee$, $\neg$, $\longrightarrow$, $\longleftrightarrow$) are written in the usual way. Function applications are written in prefix form, as in (f x_1 ... x_n). Existential quantification is denoted by $\exists\, x.$ ... and universal quantification by $\forall\, x.$ ....

We assume that the underlying theory we are defining includes the theory of ordered pairs, lists, (finite) sets, and optional data-types (all of them are built-in in Isabelle/HOL). We also assume that record data-types are available. Syntax of these operations is summarized in the first column of Fig. 4 and the semantics is informally described in the second column.

Basic types. Apart from the basic built-in types, we introduce the types used in propositional logic of CNF formulas as given by Definition 1.

Definition 1.

Variable	natural number.
Literal	either a positive variable (Pos *vbl*) or a negative variable (Neg *vbl*)
Clause	a list of literals
Formula	a list of clauses
Valuation	a list of literals

bool	the Boolean type with values True and False
ExtendedBool	the extended Boolean type with values True, False and Undef
nat	the type of natural numbers
$('a \times' b)$	the type of ordered pairs with elements of types $'a$ and $'b$
(a, b)	the ordered pair of elements a and b
$'a$ list	the type of lists with elements of type $'a$
[ ]	the empty list
$[e_1, \ldots, e_n]$	the list of n given elements $e_1, \ldots, e_n$
$e \# list$	the list obtained by prepending the element e to the list $list$
$list_1 @ list_2$	the list obtained by appending the lists $list_1$ and $list_2$
$e \in list$	e is a member of the list $list$
(removeAll e $list$)	the list obtained by removing all occurrences of the element e from the list $list$
(list_diff $list_1$ $list_2$)	the list obtained from the list $list_1$ by removing all elements of the list $list_2$ from it
(fst $list$), (hd $list$)	the first element of the list $list$
(tl $list$)	the list obtained by removing the first element of the list $list$
$list\,!\,n$	the n-th element of the list $list$
(last $list$)	the last element in the nonempty list $list$
(length $list$)	the length of the list $list$
(distinct $list$)	check if the list $list$ contains no repeating elements
(remdups $list$)	the list obtained from the list $list$ by removing all its duplicate elements
(filter P $list$)	the list obtained from the list $list$ by taking all its elements that satisfy the condition P
(map f $list$)	the list obtained from the list $list$ by applying the function f to all its elements
(prefixToElement e $list$)	the prefix of the list $list$ up to the first occurrence of the element e (including it)
$a \prec^{list} b$	the element a precedes the element b in the list $list$
$'a$ set	the type of sets with elements of type $'a$
{}	the empty set
$e \in set$	e is a member of the set set
$set_1 \cup set_2$	the set union of set_1 and set_2
$\lvert set \rvert$	the number of elements in the set set
$'a$ option	the type of optional values of the type $'a$
Some a	the optional value exists and is a
None	the optional value does not exist
$f(x := y)$	the mapping obtained from the mapping f by setting the value of x to y
$rec(\!\vert f_1 := a_1, \ldots, f_k := a_k \vert\!)$	the record obtained from the record rec by setting the values of fields $f_1, \ldots, f_k$ to values $a_1, \ldots, a_k$, respectively

Fig. 4. Summary of Isabelle's basic types and operations.

Or in Isabelle's syntax:

```
types      Variable  = nat
datatype   Literal   = Pos Variable | Neg Variable
types      Clause    = "Literal list"
types      Formula   = "Clause list"
types      Valuation = "Literal list"
```

Alternatively, (multi)sets could have been used instead of lists (e.g., valuations could have been defined as sets of literals), but we opted for lists since they more closely resemble real SAT solver implementations.

For the sake of readability, we will sometime omit printing types and use the following naming convention: literals (i.e., variables of the type Literal) are denoted by l (e.g., l, l', l_0, l_1, l_2, ...), variables by vbl, clauses by c, formulae by F, and valuations by v.

Most of the following definitions are formalized by using primitive recursion, so that they can be used to generate executable code. However, in order to simplify the presentation and improve readability we give their characterizations in an informal way and omit the Isabelle code.

Definition 2. *The opposite literal of a literal* l, denoted $\bar{l}$, is defined by: $\overline{\mathsf{Pos}\ vbl} = \mathsf{Neg}\ vbl$, $\overline{\mathsf{Neg}\ vbl} = \mathsf{Pos}\ vbl$.

We abuse the notation and overload some symbols. For example, the symbol $\in$ denotes both set membership and list membership. It is also used to denote that a literal occurs in a formula.

Definition 3. A formula F *contains a literal* l (i.e., a literal l *occurs in a formula* F), denoted $l \in F$, iff $\exists c.\ c \in F \wedge l \in c$.

The symbol vars is also overloaded and denotes the set of variables occurring in a clause, in a formula, or in a valuation.

Definition 4. The *set of variables that occur in a clause* c is denoted by $(\mathsf{vars}\ c)$. The *set of variables that occur in a formula* F is denoted $(\mathsf{vars}\ F)$. The *set of variables that occur in a valuation* v is denoted $(\mathsf{vars}\ v)$.

The semantics is introduced by the following definitions.

Definition 5. A literal l is *true in a valuation* v, denoted $v \vDash l$, iff $l \in v$. A clause c is *true in a valuation* v, denoted $v \vDash c$, iff $\exists l.\ l \in c \wedge v \vDash l$. A formula F is *true in a valuation* v, denoted $v \vDash F$, iff $\forall c.\ c \in F \Rightarrow v \vDash c$.

We will write $v \nvDash l$ to denote that l is not true in v (note that it does not mean that l is false in v), $v \nvDash c$ to denote that c is not true in v, and $v \nvDash F$ to denote that F is not true in v. We will say that l (or c, or F) is *unsatisfied* in v.

Definition 6. A literal l is *false in a valuation* v, denoted $v \vDash\neg l$, iff $\bar{l} \in v$. A clause c is *false in a valuation* v, denoted $v \vDash\neg c$, iff $\forall l.\ l \in c \Rightarrow v \vDash\neg l$. A formula F is *false in a valuation* v, denoted $v \vDash\neg F$, iff $\exists c.\ c \in F \wedge v \vDash\neg c$.

We will write $v \nvDash\neg l$ to denote that l is not false in v, $v \nvDash\neg c$ to denote that c is not false in v, and $v \nvDash\neg F$ to denote that F is not false in v. We will say that l (or c, or F) is *unfalsified* in v.

Definition 7. A valuation v is *inconsistent*, denoted $(\mathsf{inconsistent}\ v)$, iff it contains both literal and its opposite i.e., iff $\exists l.\ v \vDash l \wedge v \vDash \bar{l}$. A valuation is *consistent*, denoted $(\mathsf{consistent}\ v)$, iff it is not inconsistent.

Definition 8. A *model* of a formula F is a consistent valuation under which F is true. A formula F is *satisfiable*, denoted $(\mathsf{sat}\ F)$, iff it has a model i.e., $\exists v.\ (\mathsf{consistent}\ v) \wedge v \vDash F$.

Definition 9. A *formula* F *entails a clause* c, denoted $F \vDash c$, iff c is true in every model of F. A *formula* F *entails a literal* l, denoted $F \vDash l$, iff l is true in every model of F. A *formula* F *entails valuation* v, denoted $F \vDash v$, iff it entails all its literals i.e., $\forall l.\ l \in v \Rightarrow F \vDash l$. A *formula* F_1 *entails a formula* F_2 denoted $F_1 \vDash F_2$, if every model of F_1 is a model of F_2.

Definition 10. Formulae F_1 and F_2 are *logically equivalent*, denoted $F_1 \equiv F_2$, iff any model of F_1 is a model of F_2 and vice versa, i.e., iff $F_1 \vDash F_2$ and $F_2 \vDash F_1$.

Definition 11. A clause c is *unit* in a valuation v with a *unit literal* l, denoted $(\mathsf{isUnit}\ c\ l\ v)$ iff $l \in c$, $v \nvDash l$, $v \nvDash\neg l$ and $v \vDash\neg (c \setminus l)$ (i.e., $\forall l'.\ l' \in c \wedge l' \neq l \Rightarrow v \vDash\neg l'$).

Definition 12. A clause c is a *reason for propagation* of literal l in valuation v, denoted $(\mathsf{isReason}\ c\ l\ v)$ iff $l \in c$, $v \vDash l$, $v \vDash\neg (c \setminus l)$, and for each literal $l' \in (c \setminus l)$, the literal $\overline{l'}$ precedes l in v.

Definition 13. The *resolvent* of clauses c_1 and c_2 over the literal l, denoted $(\mathsf{resolvent}\ c_1\ c_2\ l)$ is the clause $(c_1 \setminus l)@(c_2 \setminus \bar{l})$.

Definition 14. A clause c is a *tautological clause*, denoted $(\mathsf{clauseTautology}\ c)$, if it contains both a literal and its opposite (i.e., $\exists\, l.\ l \in c \ \wedge\ \bar{l} \in c$).

Definition 15. The *conversion of a valuation* v *to a formula* $\langle v \rangle$ is the list that contains all single literal clauses made of literals from v.

Assertion trail. In order to build a non-recursive implementation of the DPLL algorithm, the notion of valuation should be slightly extended. During the solving process, the solver should keep track of the current partial valuation. In that valuation, some literals are called *decision literals*. Non-decision literals are called *implied literals*. These check-pointed sequences that represent valuations with marked decision literals will be stored in the data structure called the *assertion trail*. All literals that belong to the trail will be called *asserted literals*. The assertion trail operates as a stack and literals are always added and removed from its top. We extend the underlying theory with the type LiteralTrail, as given by Definition 16:

Definition 16.

`LiteralTrail`	a list of literals, with some of them marked as decision literals.

We will denote variables of the type LiteralTrail by M (e.g., M, M', M_0, ...).

Example 1. A trail M could be $[+1, |-2, +6, |+5, -3, +4, |-7]$. The symbol $+$ is written instead of the constructor Pos, the symbol $-$ instead of Neg and the decision literals are marked with the symbol $|$ on their left hand side.

A trail can be implemented, for example, as a list of (Literal, bool) ordered pairs and all following definitions will be based on this specific implementation. Our SAT solver implementation effectively uses the LiteralTrail datatype and so we also show its Isabelle formalization.

```
types LiteralTrail = "(Literal × bool) list"
```

Definition 17. For a trail element a, (element a) denotes the first (Literal) component and (isDecision a) denotes the second (Boolean) component. For a trail M, (elements M) (abbreviated as $\widehat{M}$) denotes the list of all its elements and (decisions M) denotes the list of all its marked elements (i.e., of all its decision literals).

```
definition element :: "(Literal × bool) ⇒ Literal"
where "element x = fst x"

definition isDecision :: "(Literal × bool) ⇒ bool"
where "isDecision x = snd x"

definition elements :: "LiteralTrail ⇒ Literal list"
where "elements M = map element M"

definition decisions :: "LiteralTrail ⇒ Literal list"
where "decisions trail = filter (λ e. isDecision e) trail"
```

Definition 18. (decisionsTo M l) is the list of all marked elements from a trail M that precede the first occurrence of the element l, including l if it is marked.

```
definition decisionsTo :: "Literal ⇒ LiteralTrail ⇒ Literal list"
where
"decisionsTo e trail = decisions (prefixToElement e trail)"
```

Example 2. For the trail given in Example 1, (decisions M) $= [-2, +5, -7]$, (decisionsTo M $+ 4$) $= [-2, +5]$, and (decisionsTo M $- 7$) $= [-2, +5, -7]$.

Definition 19. The *current level* for a trail M, denoted (currentLevel M), is the number of marked literals in M, i.e., (currentLevel M) $=$ (length (decisions M)).

```
definition currentLevel :: "LiteralTrail ⇒ nat"
where
"currentLevel trail = length (decisions trail)"
```

Definition 20. The *decision level* of a literal l in a trail M, denoted (level l M), is the number of marked literals in the trail that precede the first occurrence of l, including l if it is marked, i.e., (level l M) $=$ (length (decisionsTo M l)).

```
definition elementLevel :: "Literal ⇒ LiteralTrail ⇒ nat"
where
"elementLevel e trail = length (decisionsTo e trail)"
```

Definition 21. (prefixToLevel M *level*) is the prefix of a trail M containing all elements of M with levels less than or equal to *level*.

```
definition prefixToLevel :: "nat ⇒ LiteralTrail ⇒ LiteralTrail"
```

Example 3. For the trail in Example 1, (level $+ 1$ M) $= 0$, (level $+ 4$ M) $= 2$, (level $- 7$ M) $= 3$, (currentLevel M) $= 3$, (prefixToLevel M 1) $= [+1, |+2, +6]$.

Definition 22. The *last asserted literal of a clause* c, denoted (lastAssertedLiteral c $\widehat{M}$), is the literal from c that is in $\widehat{M}$, such that no other literal from c comes after it in $\widehat{M}$.

The function isLastAssertedLiteral is used to check if the given literal is the last asserted literal of the given clause in the given valuation.

```
definition isLastAssertedLiteral::"Literal ⇒ Literal list ⇒ Valuation ⇒ bool"
where
"isLastAssertedLiteral literal clause valuation =
   literal ∈ clause ∧ valuation ⊨ literal ∧
   (∀ literal'. literal' ∈ clause ∧ literal' ≠ literal ⟶
      literal ≺^valuation literal')"
```

The function getLastAssertedLiteral is used to detect the last asserted literal of the given clause in the given valuation.

```
definition getLastAssertedLiteral :: "Clause ⇒ Valuation ⇒ Literal"
where
"getLastAssertedLiteral clause valuation =
   last (filter (λ l. l ∈ clause) valuation)"
```

Example 4. Let c be $[+4, +6, -3]$ and M is the trail from Example 1. Then, $(\mathsf{lastAssertedLiteral}\ c\ \widehat{M}) = +4$.

4. SAT solver formalization

In this section we will present a formalized implementation of a SAT solver within the underlying theory introduced in Section 3. Different concepts and algorithms will be described in separate subsections. Together with the solver implementation we will give conditions that describe its variables and their relationships that must be invariant for the solver functions. These invariants fully characterize the role of some variables in the system and help understanding the whole system. Because invariants are listed simultaneously with the implementation, the style used can be seen as implementation driven by its specification.

Note that the following solver description is very formal and concise, and that some previous knowledge about the SAT solving technology is assumed. Useful tutorial descriptions can be found in the literature (e.g., [1,15,8,6]).

4.1. Solver state

In an imperative or object-oriented language, the state of the solver is represented by using global or class variables. Functions of the solver access and change these variables as their side-effects. In HOL, functions cannot have side-effects, so the solver state must be wrapped up in a record and passed around with each function call. Therefore, all functions in our functional implementation will receive the current solver state as their last parameter and return the modified state along with their result. However, function definitions will use monadic Haskell-style do syntax recently supported by Isabelle/HOL [2] and hide explicit state changes. For each component XXX of the state basic operations readXXX and updateXXX will be provided.

The state of the solver is represented by the following record:

```
record State =
   "getSATFlag"          :: ExtendedBool
   "getF"                :: Formula
   "getM"                :: LiteralTrail
   "getConflictFlag"     :: bool
   "getConflictClause"   :: pClause
   "getQ"                :: "Literal list"
   "getReason"           :: "Literal ⇒ pClause option"
   "getWatch1"           :: "pClause ⇒ Literal option"
   "getWatch2"           :: "pClause ⇒ Literal option"
   "getWatchList"        :: "Literal ⇒ pClause list"
   "getC"                :: Clause
   "getCl"               :: Literal
   "getCll"              :: Literal
```

The data-type pClause is just a synonym for nat and it indicates "pointers" to clauses i.e., indices of clauses in the clause list representing the formula.

Basic variables of the solver state are the following.

- The variable *SATFlag* reflects *the status of the solving process* and it remains *Undef* until the formula which is being solved is detected to be satisfiable (when *SATFlag* is set to *True*) or to be unsatisfiable (when *SATFlag* is set to *False*). Its characterization will be the main partial correctness result and it will be proved in Section 5.

  Inv[SATFlag][4]:

  $$SATFlag = True \leftrightarrow (\mathsf{sat}\ F_0)\ \wedge\ SATFlag = False \leftrightarrow \neg(\mathsf{sat}\ F_0),$$

  where F_0 is the formula tested for satisfiability.

[4] We will say that *a state satisfies an invariant* and that *an invariant holds in a state* if the components (getXXX) of the state satisfy the condition given by the invariant.

- The literal trail M contains *the current partial valuation* (i.e., $\widehat{M}$ *is* the current partial valuation). It is characterized by the following invariants:
  Inv[Mconsistent]:

  $$(\mathsf{consistent}\ \widehat{M})$$

  Inv[Mdistinct]:

  $$(\mathsf{distinct}\ \widehat{M}),$$

  which ensure that M also represents a mapping of some variables to their truth values.
  The trail M contains literals whose variables are in the initial formula F_0 and literals whose variables are in the special set of decision variables (denoted by *decisionVars* and used in decide operation formalized in Section 4.6). Note that these two sets usually coincide, but this is not necessarily the case. This *domain property* of M is given by the following invariant.
  Inv[Mvars]:

  $$(\mathsf{vars}\ M) \subseteq (\mathsf{vars}\ F_0) \cup \mathit{decisionVars}$$

- The formula F will be referred to as *the current set of clauses*. It changes during the solving process and its clauses are either (simplified) clauses of the initial formula F_0 or its consequences that are learned during the solving process. Since initial clauses are built from literals of F_0 and learned clauses are built from literals of M, the formula F satisfies the following domain property.
  Inv[Fvars]:

  $$(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0) \cup \mathit{decisionVars}$$

All clauses in F will have at least two different literals. Single literal clauses $[l]$ will never be added to F, but instead their only literal l will be immediately added to M. Indeed, adding a single literal clause $[l]$ to F would be useless because its only literal l must be contained in every satisfying valuation and $[l]$ is automatically satisfied when l is asserted. To ensure correctness, once these literals are added to M, they must never get removed from it. This is the case in the implementation we provide, since all these literals will be asserted at the decision level zero of the trail M which never gets backtracked.

As said, all clauses in F are logical consequences of F_0. Also, the decision level zero of the trail M contains literals that are logical consequences of the formula F_0. The following invariant describing the relation between the initial formula F_0, the formula F, and the trail M plays a very important role in the soundness and completeness of the solving process. It states that the formula F_0 is fully characterized by the formula F and the decision level zero of the trail M.

Inv[equivalent]:

$$F_0 \ \equiv\ F\ @\ \langle \mathsf{prefix}\widehat{\mathsf{ToLevel}}\ 0\ M\rangle$$

The fact that F contains only clauses with two or more different literals also simplifies the implementation of the two-watched literal scheme (see Section 4.4.1).

Other components of the solver state are used in specific phases of the solving process and will be explained in the following sections.

4.2. Initialization

In this section we describe the process of initializing the solver state by the given formula F_0 tested for satisfiability.

The function initialize calls addClause for each clause in F_0 which appropriately updates the solver state.

```
primrec initialize :: "Formula ⇒ State ⇒ State"
where
  "initialize [] = return ()"
| "initialize (clause # formula) =
   do
      addClause clause;
      initialize formula
   done
```

The function initialize is initially called only for initialState, so there are no decision literals in M when it is called.

```
definition initialState :: "State"
where
"initialState =
   (| getSATFlag = Undef,
      getF = [],
      getM = [],
```

```
      getConflictFlag = False,
      getConflictClause = 0,
      getQ = [],
      getWatch1 = λ c. None,
      getWatch2 = λ c. None,
      getWatchList = λ l. [],
      getReason = λ l. None,
      getC = arbitrary,
      getCl = arbitrary,
      getCll = arbitrary
   ⦈
"
```

Before we introduce the function addClause, we define an auxiliary function removeFalseLiterals used to simplify clauses. It removes all literals from the given clause that are false in the given valuation.

```
definition removeFalseLiterals :: "Clause ⇒ Valuation ⇒ Clause"
where
"removeFalseLiterals clause valuation =
   filter (λ l. valuation ⊨¬ l) clause"
```

The function addClause (called only by initialize) preprocesses the clause by removing its repeated literals and removing its literals that are false in the current trail M. After this, several cases arise.

- If the clause is satisfied in the current trail M, it is just skipped. The rationale for this is that if there is a satisfying valuation for F_0, it will be an extension of the current trail M, so it will also satisfy the clause that was skipped.
- If the clause is empty after preprocessing, the formula F_0 is unsatisfiable and *SATFlag* is set to *False*, since the empty clause cannot be satisfied in any valuation.
- Tautological clauses (i.e., clauses containing both a literal and its opposite) are also skipped since they can always be satisfied.

The two remaining cases actually update F or M.

1. As described, clauses $[l]$ containing only a single literal l are treated in a special way. Since they can only be satisfied if their literal l is true in M, l it is immediately added to M. Then a round of unit propagation (see Section 4.5) is performed, which can infer further consequences of asserting l.
2. Clauses containing more than one literal are added to F and data structures related to the two-watched literal scheme are appropriately initialized (see Section 4.4.1).

```
definition addClause :: "Clause ⇒ State ⇒ State"
where
"addClause clause =
   do
      M ← readM;
      let clause' = (remdups (removeFalseLiterals clause (elements M)));
      (if (¬ clauseTrue clause' (elements M)) then
         (if clause'=[] then
            updateSATFlag False
         else (if (length clause' = 1) then
            do
               assertLiteral (hd clause') False;
               exhaustiveUnitPropagate
            done
         else (if (¬ clauseTautology clause') then
            do
               F ← readF;
               let clauseIndex = length F;
               updateF (F @ [clause']);
               setWatch1 clauseIndex (clause' ! 0);
               setWatch2 clauseIndex (clause' ! 1)
            done
         )))
      )
   done
"
```

4.3. Top level solver operation

The only function of the solver that end-users are expected to call is the function solve. First it performs initialization and then it performs the main solver loop while the status of the solving process (given by the variable *SATFlag*) is *Undef*. The first time *SATFlag* changes, the main solver loop stops and the current value of *SATFlag* is the final solver result.

```
definition solve :: "Formula ⇒ State ⇒ State × ExtendedBool"
where
"solveFormula F0 =
   do
      initialize F0;
      solveLoop (vars F0);
      readSATFlag
   done
"

function (domintros, tailrec)
   solveLoop :: "Variable set ⇒ State ⇒ State"
where
"solve_loop decisionVars =
   do
      SATFlag ← readSATFlag;
      (if (SATFlag = Undef) then
         do
            solveLoopBody decisionVars;
            solveLoop decisionVars
         done
      )
   done
"
by pat_completeness auto
```

Note that the solve_loop is defined by general recursion, so its termination is not trivial and it will be discussed later. The body of the solver loop begins with a round of exhaustive unit propagation. After that, four different cases arise.

1. It has been detected that $M \models\neg F$. In that case we say that a *conflict* occurred.
   (a) If there are no decision literals in M, we say that a *conflict at decision level zero* occurred and it is determined that the formula F_0 is unsatisfiable. In that case, *SATFlag* is set to *False*.
   (b) If there are some decision literals in M, then the *conflict analysis and resolving procedure* is performed (see Section 4.7).
2. It has been detected that $M \not\models\neg F$.
   (a) If all variables from the fixed variable set *decisionVars* are defined in the current trail M, it is determined that the formula is satisfiable. In that case, *SATFlag* is set to *True*. The set *decisionVars* must meet additional requirements in order to guarantee the soundness of this conclusion. For example, it suffices that $(\text{vars } F_0) \subseteq$ *decisionVars*, as is the case in our implementation.
   (b) If there are some decision variables that are undefined in M, a new decision is made (see Section 4.6) and a decision literal is asserted.

The detection of clauses of F that are false in $\widehat{M}$ or unit in $\widehat{M}$ must be done efficiently so that it does not become the bottleneck of the whole solver. An optimized way to achieve this is given in Section 4.4.

```
definition solveLoopBody :: "Variable set ⇒ State ⇒ State"
where
"solveLoopBody decisionVars =
   do
      exhaustiveUnitPropagate;
      conflictFlag ← readConflictFlag;
      M ← readM;
      (if conflictFlag then
         (if (currentLevel M) = 0 then
            updateSATFlag FALSE
         else
            do
               applyConflict;
```

```
               explainUIP;
               applyLearn;
               applyBackjump
            done
         )
      else
         (if (vars (elements M) ⊇ decisionVars) then
            updateSATFlag TRUE
         else
            applyDecide decisionVars
         )
      )
   done
"
```

4.4. Conflict and unit clause detection

Each time a literal is added to M, the formula F is checked for the existence of unit or false clauses. Results of this check are stored in the following state variables.

- The variable *conflictFlag* is set when it is determined that the current set of clauses F is false in the valuation $\widehat{M}$. The invariant that fully characterizes it is:
  Inv[conflictFlagDef]:

  $$\mathit{conflictFlag} \longleftrightarrow \widehat{M} \models \neg F$$

- The number *conflictClause* is the index of a clause in F that is false in the valuation $\widehat{M}$. Its defining invariant is:
  Inv[conflictClauseDef]:

  $$\mathit{conflictFlag} \longrightarrow \mathit{conflictClause} < |F| \ \wedge \ \widehat{M} \models \neg\, (F\,!\,\mathit{conflictClause})$$

- The list Q is a list of all literals that are unit literals for clauses in F which are unit clauses with respect to the valuation $\widehat{M}$. These literals are ready to be asserted in M as a result of the *unit propagation* operation. The unit propagation queue Q is fully characterized by the following invariant.
  Inv[QDef]:

  $$\neg \mathit{conflictFlag} \longrightarrow (\forall l.\ l \in Q \longleftrightarrow (\exists c.\ c \in F \ \wedge \ (\mathsf{isUnitClause}\ c\ l\ \widehat{M})))$$

  Note that this condition guarantees the *completeness for unit propagation* i.e., it guarantees that all unit literals for unit clauses in F are contained in Q. This is not necessary for the soundness nor completeness of the whole procedure, but, if satisfied, leads to better efficiency.
  Also, there should be no repeated elements in Q.
  Inv[Qdistinct]:

  $$(\mathsf{distinct}\ Q)$$

  As Q is built of literals of F its domain (its set of variables) is included in the domain of F.
  Inv[Qvars]

  $$(\mathsf{vars}\ Q) \subseteq (\mathsf{vars}\ F_0) \cup \mathit{decisionVars}$$

- The mapping *reason* maps literals in Q to indices of clauses in F for which they are the unit literals. Since this mapping does not change when the literals from Q get asserted in M, it continues to map non-decision literals of M to indices of clauses in F that are reasons for their propagation. Notice that no reason clauses can be attached to the literals at the decision level zero. This is because literals at the decision level zero have a special role in the solving process, as they can get asserted by propagating single literal clauses which are not explicitly stored in F, as described in Section 4.1. All this is characterized by the following complex invariant.
  Inv[reasonDef]:

  $$\begin{aligned}
  &((\mathsf{currentLevel}\ M) > 0 \longrightarrow \forall l.\ l \in Q \longrightarrow \\
  &\quad (\exists c.\ (\mathit{reason}\ l) = (\mathsf{Some}\ c) \ \wedge \ c < |F| \ \wedge \ (\mathsf{isUnit}\ (F\,!\,c)\ l\ \widehat{M}))) \ \wedge \\
  &(\forall l.\ l \in \widehat{M} \ \wedge \ l \notin (\mathsf{decisions}\ M) \ \wedge \ (\mathsf{level}\ l) > 0 \longrightarrow \\
  &\quad (\exists c.\ (\mathit{reason}\ l) = (\mathsf{Some}\ c) \ \wedge \ c < |F| \ \wedge \ (\mathsf{isReason}\ (F\,!\,c)\ l\ \widehat{M})))
  \end{aligned}$$

4.4.1. Two-watched literal scheme

An efficient way to check for false and unit clauses is by using the *two-watched literal scheme*. It introduces the following variables to the state.

- Mappings $watch_1$ and $watch_2$ assign two distinguished literals to each clause of F. This condition is imposed through the following invariants.

  Inv[watchesEl]:

$$\forall c.\, c < |F| \longrightarrow \exists w1\, w2.\ (watch_1\ c) = (\mathsf{Some}\ w1) \wedge w1 \in F\,!\,c \wedge (watch_2\ c) = (\mathsf{Some}\ w2) \wedge w2 \in F\,!\,c$$

  Inv[watchesDiffer]:

$$\forall c.\, c < |F| \longrightarrow (watch_1\ c) \neq (watch_2\ c)$$

- The mapping *watchList* assigns to each literal l a list of clause indices in F that represent clauses in which l is a watched literal. This is imposed by the following invariants.

  Inv[watchListsDef]:

$$\forall l\, c.\quad c \in (watchList\ l) \longleftrightarrow c < |F| \wedge \big((watch_1\ c) = (\mathsf{Some}\ l) \vee (watch_2\ c) = (\mathsf{Some}\ l)\big)$$

  It also holds that watch lists do not contain repeated clauses.

  Inv[watchListsDistinct]:

$$\forall l.\ (\mathsf{distinct}\ (watchList\ l))$$

Next, we describe the function assertLiteral that adds the given literal (either decision or implied) to the trail M. The variables *conflictFlag*, *conflictClause*, Q, and *reason* are then updated by using the two-watched literal propagation scheme encoded by the function notifyWatches.

```
definition assertLiteral :: "Literal ⇒ bool ⇒ State ⇒ State"
where
"assertLiteral literal decision =
   do
      M ← readM;
      updateM (M @ [(literal, decision)]);
      notifyWatches (opposite literal)
   done
"
```

Before we introduce and explain the function notifyWatches, we introduce several auxiliary functions.

Functions setWatch1 and setWatch2 promote the given literal to be a new watched literal of the given clause and then add that clause to its watch list.[5]

```
definition addToWatchList :: "Literal ⇒ pClause ⇒ State ⇒ State"
"addToWatchList literal clause =
   updateWatchList literal (λ watchList. clause # watchList)
"
definition setWatch1 :: "pClause ⇒ Literal ⇒ State ⇒ State"
where
"setWatch1 clause literal =
   do
      updateWatch1 clause (Some literal);
      addToWatchList literal clause
   done
"
```

The function swapWatches swaps the two watched literals of the given clause.

[5] Only setWatch1 is listed since setWatch2 is similar.

```
definition swapWatches :: "pClause ⇒ State ⇒ State"
where
"swapWatches clause =
   do
      wa ← readWatch1 clause; wb ← readWatch2 clause;
      updateWatch1 clause wb; updateWatch2 clause wa
   done
"
```

The function getNonWatchedUnfalsifiedLiteral checks if there is a literal in the given clause, other then its watched literals, which is not false in M.

```
primrec getNonWatchedUnfalseLiteral ::
   "Clause ⇒ Literal ⇒ Literal ⇒ LiteralTrail ⇒ Literal option"
where
  "getNonWatchedUnfalseLiteral [] w1 w2 M = None"
| "getNonWatchedUnfalseLiteral (literal # clause) w1 w2 M =
      (if literal ≠ w1 ∧ literal ≠ w2 ∧ (elements M) ⊭¬ literal then
         Some literal
      else
         getNonWatchedUnfalseLiteral clause w1 w2 M
      )
   "
```

Next, we explain the essence of the two-watched literal scheme encoded in the functions notifyWatches and notifyWatchesLoop. The two-watched literal scheme relies on the fact that a watched literal of a clause can be false in $\widehat{M}$ only when the clause is either true, false or unit in M. In all other cases (when it is undefined and is not unit), both watched literals of the clause are known to be unfalsified. This is formalized by the following invariant (with two instances for $i = 1$ and $i = 2$).

$$\begin{aligned}
&\forall c.\ c < |F| \longrightarrow M \models\neg\ (watch_i\ c) \longrightarrow \\
&\quad (\exists l.\ l \in c\ \wedge\ M \models l\ \wedge\ \mathsf{level}\ l \leq \mathsf{level}\ \overline{(watch_i\ c)}) \vee \\
&\quad (\forall l.\ l \in c\ \wedge\ l \neq (watch_1\ c)\ \wedge\ l \neq (watch_2\ c) \longrightarrow \\
&\qquad M \models\neg\ l\ \wedge\ \mathsf{level}\ \bar{l} \leq \mathsf{level}\ \overline{(watch_i\ c)}).
\end{aligned}$$

Note that the additional conditions imposed on the literal levels are required only for the correctness of backjumping, as described in Section 4.7.

During the assertLiteral operation, the trail M gets extended by a literal l. When this happens, all clauses that do not have $\bar{l}$ as their watched literal still satisfy the condition of *Inv[watchDef]* and they cannot be unit nor false in the extended trail. The only clauses that could have become unit or false are the ones that have $\bar{l}$ as their watched literal. These clauses are exactly the ones whose indices are contained in $(watchList\ \bar{l})$. The function notifyWatches calls the function notifyWatchesLoop which traverses this list and processes all clauses represented by it. In order to simplify the implementation, for each processed clause index c, watches are swapped if necessary so that it is ensured that $(watch_2\ c) = \bar{l}$ and so $(watch_2\ c)$ is false. The following cases may further arise:

1. If it can be quickly detected that the clause $F\,!\,c$ contains a true literal t, there is no need to change its watches, since it satisfies the condition of *Inv[watchDef]* for the extended trail. In order to achieve high performance, this check should be done only by using the clause index and other data structures which are most of the time present in the processor cache, without accessing the clause itself. The older solvers checked only if $(watch_1\ c)$ is true in M and this is the case in the implementation we provide. Some new solvers sometimes cache some arbitrary literals of the clause and check if they are true in M.
2. If a quick check does not detect a true literal t, then the clause is accessed and its other literals are examined by the function getUnfalsifiedNonWatchedLiteral.
   (a) If there exists a non-watched literal l that is not false in M, it becomes a new $(watch_2\ c)$.
   (b) If all non-watched literals and $(watch_1\ c)$ are false in M, then the whole clause is false and *conflictFlag* is raised. The watches are not changed, since they will both become undefined in M, if the backjump operation is performed (see Section 4.7).
   (c) If all non-watched literals are false in M, but $(watch_1\ c)$ is undefined, then the clause just became a unit clause and $(watch_1\ c)$ is enqueued in Q for propagation (if it is not already present there). The reason for its propagation is set to c. The watches are not changed, as the clause will have a true literal $(watch_1\ c)$ after propagation.

When a literal which was not watched becomes a new $(watch_2\ c)$, the literal $\bar{l}$ stops being the watched literal of c and the clause index c should be removed from its watch list. Since this happens many time during the traversal performed by the

notifyWatchesLoop, it turns out that it is more efficient to regenerate the new watch list for the literal $\bar{l}$, then to do successive remove operations instead. This is the role of the *newWl* parameter in the notifyWatchesLoop function.

```
definition notifyWatches :: "Literal ⇒ State ⇒ State"
where
"notifyWatches literal =
   do
      wl ← readWatchList literal;
      notifyWatchesLoop literal wl []
   done
"

primrec notifyWatchesLoop ::
  "Literal ⇒ pClause list ⇒ pClause list ⇒ State ⇒ State"
where
  "notifyWatchesLoop literal [] newWl =
     updateWatchList literal (λ wl. newWl)"
| "notifyWatchesLoop literal (clause # list') newWl =
  do
     w1' ← readWatch1 clause; w2' ← readWatch2 clause;
     if (Some literal = w1') then (swapWatches clause);
     (case w1' of Some w1 ⇒ (case w2' of Some w2 ⇒ (
        do
           M ← readM;
           (if literalTrue w1 (elements M) then
              notifyWatchesLoop literal list' (clause # newWl)
           else
              do
                 F ← readF;
                 let ul = getNonWatchedUnfalseLiteral (F!clause) w1 w2 M in
                 (case ul of
                   Some l' ⇒
                    do
                       setWatch2 clause l';
                       notifyWatchesLoop literal list' newWl
                    done
                 | None ⇒
                    (if (literalFalse w1 (elements M)) then
                       do
                          updateConflictFlag True;
                          updateConflictClause clause;
                          notifyWatchesLoop literal list' (clause # newWl)
                       done
                    else
                       do
                          Q ← readQ;
                          if (¬ w1 el Q) then (updateQ (Q @ [w1]));
                          updateReason w1 clause;
                          notifyWatchesLoop literal list' (clause # newWl)
                       done
                    )
                 )
              done
           )
        done
     )))
  done
"
```

The invariants *Inv[watchListsDef]* and *Inv[watchesEl]* together guarantee that for each clause there will always be two watched literals (hence, the missing None branches in the case expressions are indeed not needed).

4.5. Unit propagation

The operation of unit propagation asserts unit literals of unit clauses of F. Since the two-watched literal scheme is complete for false and unit clause detection (as the function assertLiteral preserves *Inv[conflictFlagDef]* and *Inv[QDef]*), all unit literals of clauses in F can be found in Q. This makes unit propagation a rather trivial operation — literals are picked from Q and asserted until Q is emptied or until a conflict is detected.

```
definition applyUnitPropagate :: "State ⇒ State"
where
"applyUnitPropagate =
   do
      Q ← readQ; assertLiteral (hd Q) False;
      Q' ← readQ; updateQ (tl Q')
   done
"

function (domintros, tailrec) exhaustiveUnitPropagate :: "State ⇒ State"
where
"exhaustiveUnitPropagate =
   do
      conflictFlag ← readConflictFlag; Q ← readQ;
      (if (¬ conflictFlag ∧ Q ≠ []) then
         do
            applyUnitPropagate;
            exhaustiveUnitPropagate
         done
      )
   done
"
by pat_completeness auto
```

Notice that the termination of the exhaustiveUnitPropagate function is non-trivial, since it is defined by using the general recursion and it will be discussed later.

4.6. Decision heuristics

When unit propagation is exhausted, no new literal can be inferred and a kind of backtracking search must be performed. This search is driven by the guesses made by the *decision heuristic*. The heart of the decision heuristic is the selectLiteral function whose role is to pick a literal whose variable is in the fixed set of decision variables *decisionVars*, but which is not yet asserted in M. The literals are selected based on some given criteria. Many different criteria can be used and experimental evidence shows that this heuristic is often crucial for a solver's performance. However, in this paper we will specify it only by its effect given by the following postcondition.

```
consts selectLiteral :: "Variable set ⇒ State ⇒ Literal × State"
axioms selectLiteral_def:
"let diff = decisionVars \ vars (elements (getM state)) in
   diff ≠ ∅ ⟶ var (selectLiteral decisionVars state) ∈ diff"

definition applyDecide :: "Variable set ⇒ State ⇒ State"
where
"applyDecide decisionVars =
   do
      l ← selectLiteral decisionVars;
      assertLiteral l True
   done
"
```

4.7. Conflict handling

The conflict handling procedure consists of the *conflict analysis*, *learning* and *backjumping* and it is executed whenever a conflict occurs at a decision level higher then zero (when the conflict occurs at decision level zero, then the formula is determined to be unsatisfiable). After the conflict handling procedure, a top portion of trail is removed and a non-conflicting

state is restored. Unlike the classic backtrack operation which would remove only the last decision made, the backjump operation performs a form of non-chronological backtracking which undoes as many decisions as possible. Backjumping is guided by a *backjump clause*, which is a consequence of the formula F_0 and which corresponds to the variable assignment that led to the conflict. Backjump clauses are constructed in the process called *conflict analysis* as described in Section 4.7.1. When the backjump clause is constructed, the top literals from the trail M are removed until the backjump clause becomes a unit clause in M. From that point, its unit literal is propagated and the search process continues.

Several components of the solver state are used during the conflict handling procedure.

- The clause C represents the current conflict analysis clause, which becomes the backjump clause once the conflict analysis process is finished. This clause is characterized by the following invariants.[6]

  Inv[CFalse]:

  $$\mathit{conflictFlag} \longrightarrow M \models \neg\, C$$

  Inv[CEntailed]:

  $$\mathit{conflictFlag} \longrightarrow F_0 \models C$$

The following variables represent different aspects of the clause C and are cached in the solver state only for performance reasons.

- The literal C_l is the last asserted literal of $\overline{C}$ in the trail M.

  Inv[ClDef]:

  $$\mathit{conflictFlag} \longrightarrow (\mathsf{isLastAssertedLiteral}\ C_l\ \overline{C}\ M)$$

- The literal C_{ll} is the last asserted literal of $\overline{C} \setminus C_l$.

  Inv[CllDef]:

  $$\mathit{conflictFlag} \wedge \overline{C} \setminus C_l \neq [] \longrightarrow (\mathsf{isLastAssertedLiteral}\ C_{ll}\ (\overline{C} \setminus C_l)\ M)$$

- The number C_n is the number of literals on the highest decision level of the trail M.

  Inv[CnDef]:

  $$\mathit{conflictFlag} \longrightarrow$$
  $$C_n = (\mathsf{length}\ (\mathsf{filter}\ (\lambda\, l.\ \mathsf{level}\ \bar{l}\ M = \mathsf{currentLevel}\ M)\ (\mathsf{remdups}\ C)))$$

4.7.1. Conflict analysis

In order to implement the conflict analysis procedure, we introduce several auxiliary functions.

The function findLastAssertedLiteral is used to set the value of C_l based on the current values of C and M.

```
definition findLastAssertedLiteral :: "State ⇒ State"
where
"findLastAssertedLiteral =
   do
      C ← readC; M ← readM;
      updateCl (getLastAssertedLiteral (opposite C) (elements M))
   done
"
```

The function countCurrentLevelLiterals is used to set the value of C_n based on the current values of C and M.

```
definition countCurrentLevelLiterals :: "State ⇒ State"
where
"countCurrentLevelLiterals =
   do
      M ← readM; C ← readC;
      let cl = currentLevel M;
      let cll = filter (λ l. elementLevel (opposite l) M = cl) C;
      updateCn (length cll)
   done
"
```

[6] All invariants that are relevant for the conflict handling process need to hold only until the conflict has been resolved. Therefore, they are guarded by the condition *conflictFlag* $\longrightarrow$ so that they can be treated as other global invariants.

Since for some literals asserted at the decision level zero there are no reason clauses in F, it is required that the clause C does not contain literals from the decision level zero. Also, it is reasonable to require that the clause C does not contain repeated literals. The function setConflictAnalysisClause sets the clause C to the given one, but first it preprocesses it by removing duplicates and literals asserted at decision level zero. It also caches the values of C_l and C_n.

```
definition setConflictAnalysisClause :: "State ⇒ State"
where
"setConflictAnalysisClause clause =
   do
      M ← readM;
      let oppM0 = oppositeLiteralList (elements (prefixToLevel 0 M));
      updateC (remdups (list_diff clause oppM0));
      findLastAssertedLiteral; countCurrentLevelLiterals
   done
"
```

The conflict analysis algorithm can be described as follows:

- The conflict analysis process starts with a conflict clause itself (the clause of F that is false in M) and the clause C is initialized to it. The function applyConflict initializes the clause C to the current conflict clause.

```
definition applyConflict :: "State ⇒ State"
where
"applyConflict =
   do
      F ← readF; conflictClause ← readConflictClause;
      setConflictAnalysisClause (F ! conflictClause)
   done
"
```

- Each literal contained in the current clause C is false in the current trail M and is either a decision made by the search procedure or the result of some propagation. For each propagated literal l, there is a clause c that caused the propagation. These clauses are called *reason clauses* and (isReason $c\ l\ \widehat{M}$) holds. Propagated literals from the current clause C are then replaced (we say *explained*) by other literals from the reason clauses, continuing the analysis backwards. The explanation step can be seen as a resolution between the backjump and the reason clause. The function applyExplain performs this resolution.

```
definition applyExplain :: "Literal ⇒ State ⇒ State"
where
"applyExplain literal =
   do
      reason' ← readReason literal;
      (case reason' of Some reason ⇒
         do
            C ← readC; F ← readF;
            let res = resolve C (nth F reason) (opposite literal);
            setConflictAnalysisClause res
         done
      )
   done
"
```

Notice that *Inv[reasonDef]* guarantees that each propagated literal has an assigned reason clause and that the missing None branch in the case expression is not necessary.

- The conflict analysis procedure we implemented always explains the last asserted literal of $\overline{C}$ and the procedure is repeated until the isUIP condition is fulfilled, i.e., until there is exactly one literal in $\overline{C}$ such that all other literals of $\overline{C}$ are asserted at strictly lower decision levels. This condition can be easily checked by examining the value of C_n. The implementation of this technique is given by the function applyExplainUIP.

```
function (domintros, tailrec) explainUIP :: "State ⇒ State"
where
"explainUIP =
   do
      Cn ← readCn;
```

```
      (if (Cn ≠ 1) then
         do
            Cl ← readCl; applyExplain Cl;
            explainUIP
         done
      )
   done
"
by pat_completeness auto
```

Notice that this function is defined by general recursion so its termination is non-trivial and it will be discussed later.

4.7.2. Learning

During the learning process, redundant clauses that are logical consequences of the initial formula F_0 are learned. Learned clauses containing multiple literals are added to the F, while single literal clauses extend the level zero of the trail M. In our implementation (as is often the case in modern SAT solvers), the only clauses that are being learned are the backjump clauses. Since we require that all clauses in F have more than two different literals, if a backjump clause C contains only one literal, then learning is not explicitly performed (it is performed implicitly as a part of the backjumping operation). The implementation of learning is given by the function applyLearn. After extending F by C, the watch literals for the clause C are set in a way which ensures *Inv[watchDef]*. At the same time, the literal C_{ll} is computed and cached.

```
definition applyLearn :: "State ⇒ State"
where
"applyLearn =
   do
      C ← readC; Cl ← readCl;
      (if (C ≠ [opposite Cl]) then
         do
            F ← readF; M ← readM;
            updateF (F @ [C]);
            let l = Cl;
            let ll = getLastAssertedLiteral (removeAll l (opposite C)) (elements M);
            let clauseIndex = length F;
            setWatch1 clauseIndex (opposite l);
            setWatch2 clauseIndex (opposite ll);
            updateCll ll
         done
      )
   done
"
```

4.7.3. Backjumping

The backjump operation consists of removing literals from M up to a minimal level in which the backjump clause C becomes a unit clause, after which its unit literal $\overline{C_l}$ is propagated. This level is found by using the function getBackjumpLevel.

```
definition getBackjumpLevel :: "State ⇒ nat × State"
where
"getBackjumpLevel =
   do
      C ← readC; Cl ← readCl;
      (if C = [opposite Cl] then
         return 0
      else
         do
            Cll ← readCll; M ← readM;
            return (elementLevel Cll M)
         done
      )
   done
"
```

The function applyBackjump performs the backjump operation itself.

```
definition applyBackjump :: "State ⇒ State"
where
"applyBackjump =
   do
      level ← getBackjumpLevel; Cl ← readCl; M ← readM; F ← readF;
      updateConflictFlag False;
      updateQ [];
      updateM (prefixToLevel level M);
      if (level > 0) then updateReason (opposite Cl) (length F - 1);
      assertLiteral (opposite Cl) False
   done
"
```

Notice that after taking the prefix of M, it is concluded that conflict has been successfully resolved (so *conflictFlag* is unset), and that there are no unit clauses in F with respect to the taken prefix of M (so Q is cleared). For these conclusions to be valid, it is required that no new decisions are made once M is in a conflicting state. Also, unit propagation has to be exhaustive and no new decisions should be made while there are unit clauses in F. These conditions are imposed by the following invariants.

Inv[noDecisionsWhenConflict]:

$$\forall level' < (\text{currentLevel } M) \longrightarrow (\text{prefixToLevel } level' M) \not\models \neg F$$

Inv[noDecisionsWhenUnit]:

$$\forall level' < (\text{currentLevel } M) \longrightarrow \neg\exists c\ l.\ c \in F \wedge (\text{isUnitClause } c\ l\ (\text{prefixToLevel } level' M))$$

5. Highlights of the total correctness proof

The invariants listed in Section 4 are sufficient to prove the total correctness of the procedure. Proving that they are preserved by all solver functions was the most involved part of the total correctness proof. These proofs are available [14] and we will not list them here.

Next we will describe the techniques used to prove the termination of our main solver function solve. We will also prove its total correctness theorem.

5.1. Termination

In the code presented in this paper, only the functions exhaustiveUnitPropagate, explainUIP, and solveLoop are defined by using general recursion and it is not obvious if they are terminating. The only function that end-users of the solver are expected to call directly is the function solve as it is the solver's only entry-point. This means that all three functions defined by general recursion are called only indirectly by the function solve and all parameters that are passed to them are computed by the solver. Therefore, these functions can be regarded as partial functions and it is not necessary to show that they terminate for all possible values of their input parameters. It suffices to show that they terminate for those values of their input parameters that could actually be passed to them during a solver's execution starting from the initial state.

We use Isabelle's built-in features to model this kind of partiality [12].

1. Notice that all three functions are defined by using the tail recursion and annotated by the directive `tailrec`. This is a very important feature, and Isabelle can accept these functions whether they terminate or not. However, in order to generate executable code, the (partial) termination of these functions must be shown.
2. When an n-ary function f is defined by using a general recursion, a predicate f_dom which tests if an n-tuple $(a_1, \ldots, a_n)$ is in the domain of f (i.e., if f terminates on input $(a_1, \ldots, a_n)$) is automatically generated. If the function definition is annotated by the directive `domintros`, Isabelle generates a theorem of the form

   $$g \longrightarrow (\text{f\_dom } (f_1(a_1), \ldots, f_n(a_n))) \longrightarrow (\text{f\_dom } (a_1, \ldots, a_n)),$$

   for each recursive call $\text{f}(f_1(a_1), \ldots, f_n(a_n))$ in the definition of f, where g is a guard for this recursive call. Until the termination of f is proved i.e., until f is proved to be total, the usual induction scheme theorem for the function f (which would be called `f.induct`) cannot be proved and used. However, when f is defined a weaker, partial induction scheme theorem (called `f.pinduct`) is automatically proved. It differs from the usual induction scheme only because it adds the domain predicate f_dom both to the induction base and to the induction steps. These domain predicates are then carried over and assumed in all lemmas about the function f which are proved by (the partial) induction. Still, in order to complete the whole correctness proof, at one point they have to be discharged. This is done by proving that all inputs passed to the function f imply the domain predicate.

In our case, we know that invariants are preserved throughout any solver's run and that each state for which our solver functions are called satisfies all given invariants. We show that some of these invariants imply the domain predicates, i.e., that our three functions defined by general recursion terminate for states in which these invariants hold.

As an illustration, we will outline the proof that the function `exhaustiveUnitPropagate` (p4348) terminates if its input satisfies certain invariants.

In order to prove this, we introduce a well-founded ordering of trails such that applications of applyUnitPropagate advances, i.e., decreases the trail, in that ordering. So, let us first define an ordering $\succ^{lit}$ of marked literals (it is trivially well-founded).

Definition 23. $l_1 \succ^{lit} l_2 \longleftrightarrow (\text{isDecision } l_1) \wedge \neg(\text{isDecision } l_2)$

Now we can introduce an ordering of trails, which will be used as a basis for the ordering that we are constructing.

Definition 24.

$$M_1 \succ_{trail} M_2 \longleftrightarrow M_1 \succ^{lit}_{lex} M_2,$$

where $\succ^{lit}_{lex}$ is a lexicographic extension of relation $\succ^{lit}$.

The function applyUnitPropagate decreases the trail in this ordering (trivially, by the definition of lexicographic extension), but, unfortunately, this ordering need not be well-founded. However, since invariants hold in every state during the solver's operation, we can make a restriction of $\succ_{trail}$ that is also well-founded.

Definition 25.

$$\begin{aligned} M_1 \succ^{r}_{trail} M_2 \longleftrightarrow\ & (\text{distinct } M_1) \wedge (\text{vars } M_1) \subseteq \mathit{Vbl} \\ & (\text{distinct } M_2) \wedge (\text{vars } M_2) \subseteq \mathit{Vbl} \wedge \\ & M_1 \succ_{trail} M_2 \end{aligned}$$

This is the ordering we were looking for and now we can prove a lemma saying that if the state satisfies certain invariants, then it is in the domain of the applyUnitPropagate function (i.e., that this function terminates when applied to that state).

Lemma 1. *If the set decisionVars is finite and the state s is such that:*

(a) *Inv[Mconsistent] (p4341) and Inv[Mdistinct] (p4341) hold in s,*
(b) *Inv[Mvars] (p4341), Inv[Fvars] (p4341), and Inv[Qvars] (p4344) hold in s,*
(c) *Inv[conflictFlagDef] (p4344), Inv[QDef] (p4344), and Inv[Qdistinct] (p4344) hold in s,*
(d) *Inv[watchListsDef] (p4345) and Inv[watchListsDistinct] (p4345) hold in s,*
(e) *Inv[watchesEl] (p4345), Inv[watchesDiffer] (p4345) and Inv[watchDef] (p4346) hold in s,*

then the function exhaustiveUnitPropagate *terminates when applied to s, i.e.,* (exhaustiveUnitPropagate_dom *s*).

Proof. If Q is empty or *conflictFlag* is raised in the state s, then the function exhaustiveUnitPropagate terminates and s is trivially in its domain. So, let us assume that Q is not empty and *conflictFlag* is false.

The proof is carried out by well-founded induction on the ordering $\succ^{r}_{trail}$. Assume, as an inductive hypothesis, that the statement holds for all states s' for which $s \succ^{r}_{trail} s'$. Let $s' = (\text{applyUnitPropagate } s)$. Since invariants hold in s and are preserved by the applyUnitPropagate function, they hold in s' as well.[7] Since the trail M in s' is extended by a single literal, by the definition of $\succ^{r}_{trail}$, it holds that the $s \succ^{r}_{trail} s'$. So, by inductive hypothesis, it holds that (exhaustiveUnitPropagate_dom s'). The lemma then follows from the domain introduction theorem `exhaustiveUnitPropagate.domintros`:

$$\begin{aligned} & \neg \mathit{conflictFlag}_s \wedge Q_s \neq [] \longrightarrow \\ & \quad (\text{exhaustiveUnitPropagate\_dom } (\text{applyUnitPropagate } s)) \longrightarrow \\ & \quad\quad (\text{exhaustiveUnitPropagate\_dom } s). \quad \square \end{aligned}$$

Termination (on relevant inputs) of the explainUIP and solveLoop functions is proved in a similar way. The termination proof for the solveLoop function uses the same ordering $\succ^{r}_{trail}$ and the termination proof for explainUIP uses the following well-founded ordering of clauses $\succ^{M}_{clause}$ parametrized by the trail M.

Definition 26.

$$C_1 \succ^{M}_{clause} C_2 \longleftrightarrow \{\text{remdups } \overline{C_1}\} \succ^{\widehat{M}}_{mult} \{\text{remdups } \overline{C_2}\},$$

where $\{\ldots\}$ denotes the multiset of list elements and $\succ^{\widehat{M}}_{mult}$ is the multiset extension of the ordering $\succ^{\widehat{M}}$ induced by the list $\widehat{M}$ ($e_1 \succ^{\widehat{M}} e_2$ iff e_1 occurs after e_2 in the list $\widehat{M}$).

[7] Note that only *Inv[distinctM]*, and *Inv[varsM]* need to hold in order to use the ordering $\succ^{r}_{trail}$. However, we had to assume many additional invariants in the premises of this lemma, because they are needed to show that these three key invariants are preserved when applyUnitPropagate is applied.

5.2. Total correctness

Total correctness of the `solve` function is given by the following theorem.

Theorem 1.

$$((\text{solve } F_0) = \mathit{True} \wedge (\text{sat } F_0)) \vee ((\text{solve } F_0) = \mathit{False} \wedge \neg(\text{sat } F_0))$$

Assuming that all invariants hold in each state reached during the solve function execution, the proof of Theorem 1 relies on the following two soundness lemmas, which correspond to the two places in the solver code where *SATFlag* is changed.

Lemma 2. *If in some state s it holds that:*

(a) *Inv[equivalent] (p4341) holds in s,*
(b) *Inv[conflictFlagDef] (p4344) holds in s,*
(c) *conflictFlag is true in s,*
(d) $(\text{currentLevel } M) = 0$ *in s,*

then it holds that $\neg(\text{sat } F_0)$.

Proof. From $(\text{currentLevel } M) = 0$ it follows that $(\text{prefixToLevel } 0\ M) = M$. Hence, from *Inv[equivalent]* it follows that $F \mathbin{@} \langle \widehat{M} \rangle \equiv F_0$. Since from *conflictFlag* and *Inv[conflictFlagDef]* it holds that $M \models\neg F$, by monotonicity it also holds that $M \models\neg F \mathbin{@} \langle \widehat{M} \rangle$. Since $F \mathbin{@} \langle \widehat{M} \rangle \models M$, the formula $F \mathbin{@} \langle \widehat{M} \rangle$ is false in a valuation that it entails, so it is unsatisfiable. Since F_0 is logically equivalent to $F \mathbin{@} \langle \widehat{M} \rangle$, it is also unsatisfiable. □

Lemma 3. *If in some state s it holds that:*

(a) $(\text{vars } F_0) \subseteq$ *decisionVars,*
(b) *Inv[Mconsistent] (p4341) holds in s,*
(c) *Inv[Fvars] (p4341) holds in s,*
(d) *Inv[equivalent] (p4341) holds in s,*
(e) *Inv[conflictFlagDef] (p4344) holds in s,*
(f) *conflictFlag is false in s,*
(g) $(\text{vars } \widehat{M}) \supseteq$ *decisionVars in s,*

then $(\text{sat } F_0)$ *and* $(\text{model } \widehat{M}\ F_0)$ *hold.*

Proof. From *Inv[Fvars]*, it follows that $(\text{vars } F) \subseteq (\text{vars } F_0) \cup$ *decisionVars*. With $(\text{vars } F_0) \subseteq$ *decisionVars*, it holds that $(\text{vars } F) \subseteq$ *decisionVars*. With $(\text{vars } \widehat{M}) \supseteq$ *decisionVars*, it holds that $(\text{vars } F) \subseteq (\text{vars } \widehat{M})$ and $\widehat{M}$ is a total valuation with respect to the variables of F. Therefore, it is either the case that $\widehat{M} \models\neg F$ or $\widehat{M} \models F$. Since *conflictFlag* is false, by *Inv[conflictFlagDef]* it holds that $\widehat{M} \not\models\neg F$, so it must be the case that $\widehat{M} \models F$. It trivially holds that $\widehat{M} \models \langle \widehat{\text{prefixToLevel}}\ 0\ M \rangle$ and $\widehat{M}$ is consistent by *Inv[Mconsistent]*. Therefore $\widehat{M}$ is a model for $F \mathbin{@} \langle \widehat{\text{prefixToLevel}}\ 0\ M \rangle$. Since $F_0 \equiv F \mathbin{@} \langle \widehat{\text{prefixToLevel}}\ 0\ M \rangle$, it holds that $\widehat{M}$ is also a model for F_0 and $(\text{sat } F_0)$ holds. □

6. Discussion on proof management

Although it is hard to quantify the efforts invested in this work, we can estimate it to be around one man-year. The proof scripts are around 25 000 lines of Isabelle code and the generated PDF proof documents are around 700 pages long. These numbers are of course heavily dependent on the indentation style used. Proof-checking time by Isabelle is under 5 min on a 1.6 GHz/512 MB RAM machine running Linux. We estimate that careful investigation of the proof text and its reorganization mainly by extracting some common parts of different proofs into lemmas could lead to 10%–20% reductions.

During this verification effort some interesting technical issues arose. In order to make such a large-scale verification effort possible, it was necessary to introduce some kind of modularity to the formalization. The crucial step in this direction was to prove the properties of abstract state transition systems for SAT [20,11] and then use these proofs in the correctness proof of the low-level implementation presented here. A good direction to follow would be to define internal data-structures (for example the assertion trail) as abstract data-types (ADT) with some desired properties given axiomatically. Although, unfortunately, this has not been explicitly done in our formalization, this idea has been followed to some extent. Namely, after introducing basic definitions, we showed lemmas that could be regarded as axioms of the ADT and all further proofs relied only on those lemmas, without using the low-level properties of the implementation. This, of course, enables changing the low-level implementation into a more efficient one without changing much of the whole correctness proof. We think that explicit encoding of the ADT approach (for example by using type-classes or locales [21]) would lead to even more flexible formalization and is a step in the right direction.

When proving properties about recursively defined functions we had a dilemma whether to repeat the same induction scheme in proofs of many similar lemmas (one for each property of the recursive function) or to formulate one bigger lemma

that groups all assumptions and conclusions for several properties that are being shown. We took the second approach and reduced the total number of lemmas and the total size of proofs, but the price that had to be payed is that we lost track of which assumptions are effectively used for proving a specific conclusion. For example, most of our high-level lemmas that show that invariants are preserved by the function calls assume that all invariants hold before the function call and show that all invariants hold after the function call. The only way to find out which invariants are necessary to hold before the function call so that a specific invariant holds is after it is reading the proof texts which can be a tedious task.

7. Related work

First steps towards verification of SAT and related SMT solvers have been recently made. Shankar has mechanically proved soundness, completeness, and decidability of propositional logic (by means of a satisfiability solver) [22]. Zhang and Malik have informally proved correctness of a modern SAT solver [26]. Abstract state transition systems [20,11] describe high-level operation of modern SAT solvers and the authors have informally proved their correctness. Marić and Janičić have mechanically verified the classic DPLL procedure by shallow embedding into Isabelle/HOL [16]. Lescuyer and Conchon have mechanically verified a classic DPLL based SAT solver within the system Coq [13]. Shankar and Vaucher have mechanically verified a higher level description of a modern DPLL procedure within the system PVS. Marić has previously given [15] a tutorial exposure of the modern SAT solving techniques (both high and low level) with correctness properties formulated in a Hoare-style framework and proved (to some extent) mechanically within the system Isabelle.

To the best of our knowledge, this is the first paper that presents a full mechanical proof of the total correctness (soundness, termination and completeness) of a SAT solver implementation that covers both modern high-level SAT solving algorithms (e.g., conflict-driven learning) and low-level implementation techniques (most notably the two-watched literal propagation scheme).

8. Further work

The specification of the SAT solver given in this paper is such that a fully executable code in a functional language can be automatically generated from it, providing that an executable decision heuristic is supplied. However, the efficiency of the generated code must still be improved, if we want to get a competitive solver.

First, there are several low-level algorithmic improvements that have to be made. For example, in the current implementation, checking if a literal is true in a trail M requires performing a linear-time scan through the list. Most real-world solvers cache truth values of all literals in an array and so allow a constant time check. Also, the conflict analysis phase is expressed here in a bit more abstract way then in implementations of MiniSat style solvers.

Next, some higher-level heuristics have to be implemented more carefully. For example, we have only made tests with a trivial decision heuristic that selects a random undefined literal, but in order to have a usable solver, a more involved decision heuristic (e.g., the MiniSat one) should be used. It would also be useful to implement forgetting and restarting techniques [11,20].

Although these modifications require us to invest more work, we believe that they are straightforward. However, the most problematic issue is the fact that because of the pure functional nature of HOL no side-effects are possible and there can be no *destructive updates* of data-structures. It is possible to adapt the code generator to generate monadic Haskell and imperative ML code which would lead to huge efficiency benefits since it allows mutable references and arrays. We hope that with these modifications, the generated code could become comparable to real-world SAT solvers and this would be the main direction of our further work.

9. Conclusions

In this paper, we have presented a formalization and a total correctness proof of a MiniSAT-like SAT solver within the system Isabelle/HOL. The solver is based on the DPLL procedure and employs most state-of-the-art SAT solving techniques including the conflict-guided backjumping, clause learning and the two-watched unit propagation scheme. The described solver specification can serve as a basis for implementation of an efficient and correct SAT solver. One possible approach for that would be to manually implement a SAT solver (in an imperative programming language) by strictly following the descriptions of the solver given in this paper. However, the highest possible level of trust could be achieved only if fully executable code (in a functional programming language) is automatically generated by using the Isabelle's built-in code generator. Although this has been done, the efficiency of generated code should further be improved and that is the field of our future research. We hope that this work can facilitate a better understanding of modern SAT solvers. The final product of this research will be a trusted and efficient SAT solver that can be used either independently or as a kernel for checking results of other untrusted verifiers. We also hope that this work shows that, thanks to recent advances in both automated and semi-automated software verification technology, it is possible to have a fully verified implementation of a very non-trivial software system.

Acknowledgements

The author wishes to thank Prof. Predrag Janičić for helpful discussions and valuable assistance in writing this paper, and to thank Prof. Natarajan Shankar for sharing his unpublished manuscript. I also want to thank an anonymous TCS reviewer for careful reading of the earlier version of the text and giving valuable corrections and suggestions.

References

[1] A. Biere, M. Heule, H. van Maaren, T. Walsh, Handbook of Satisfiability, IOS Press, Amsterdam, 2009.
[2] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkok, J. Matthews, Imperative functional programming with Isabelle/HOL, in: TPHOLs '08, Montreal, 2008.
[3] S.A. Cook, The complexity of theorem-proving procedures, in: 3rd STOC, New York, 1971, pp. 151–158.
[4] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Communications of the ACM 5 (7) (1962) 394–397.
[5] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of the ACM 7 (3) (1960) 201–215.
[6] N. Een, N. Sorensson, An extensible SAT solver, in: SAT'03, in: LNCS, vol. 2919, S. Margherita Ligure, 2003, pp. 502–518.
[7] A. van Gelder, Verifying propositional unsatisfiability: pitfalls to avoid, in: SAT '07, in: LNCS, vol. 4501, Lisbon, 2007, pp. 328–333.
[8] C.P. Gomes, H. Kautz, A. Sabharwal, B. Selman, Satisfiability solvers, in: Handbook of Knowledge Representation, Elsevier, 2007.
[9] E. Goldberg, Y. Novikov, Berkmin: a rast and robust SAT solver, in: DATE'02, Paris, 2002, pp. 142–149.
[10] F. Haftmann, Code generation from Isabelle/HOL theories. http://isabelle.in.tum.de/documentation.html, 2008.
[11] S. Krstić, A. Goel, Architecting solvers for SAT modulo theories: nelson-oppen with DPLL, in: FroCos'07, in: LNCS, vol. 4720, Liverpool, 2007, pp. 1–27.
[12] A. Krauss, Defining recursive functions in Isabelle/HOL. http://isabelle.in.tum.de/documentation.html, 2008.
[13] S. Lescuyer, S. Conchon, A reflexive formalization of a SAT solver in coq, in: TPHOLs'08: Emerging Trends, Montreal, 2008.
[14] F. Marić, Formal verification of modern SAT solvers, The Archive of Formal Proofs, 2008. http://afp.sf.net/entries/SATSolverVerification.shtml.
[15] F. Marić, Formalization and implementation of SAT solvers, Journal of Automated Reasoning 43 (1) (2009) 81–119.
[16] F. Marić, P. Janičić, Formal correctness proof for DPLL procedure, Informatica 21 (1) (2010) 57–78.
[17] F. Marić, P. Janičić, SAT verification project, in: TPHOLs'09: Emerging Trends, Munich, 2009.
[18] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: DAC'01, Las Vegas, 2001, pp. 530–535.
[19] J P. Marques-Silva, K A. Sakallah, Grasp: A Search Algorithm for Propositional Satisfiability, IEEE Transactions on Computers 48 (5) (1999) 506–521.
[20] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T), Journal of the ACM 53 (6) (2006) 937–977.
[21] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, in: LNCS, vol. 2283, Springer, 2002.
[22] N. Shankar, Towards Mechanical Metamathematics, Journal of Automated Reasoning 1 (4) (1985) 407–434.
[23] N. Shankar, M. Vaucher, The mechanical verification of a DPLL-based satisfiability solver, Personal correspondence.
[24] H. Zhang, SATO: An efficient propositional prover, in: CADE-14, in: LNCS, vol. 1249, Townsville, 1997, pp. 272–275.
[25] Verified software: theories, tools, experiments, Conference. http://vstte.ethz.ch/.
[26] L. Zhang, S. Malik, validating SAT solvers using independent resolution-based checker, in: DATE '03, Washington DC, 2003, p. 10880.